ESTRUTURAS DE DADOS E ALGORITMOS

Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 5
ALGORITMOS BÁSICOS DE PESQUISA

OBJETIVOS

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o método de pesquisa sequencial de dados;
- Compreender o algoritmo de pesquisa binária;
- Conhecer a diferença, em termos de desempenho, entre uma pesquisa binária e uma sequencial;
- Saber implementar em C++ o algoritmo de pesquisa sequencial numa tabela e num ficheiro;
- Saber implementar em C++ o algoritmo de pesquisa binária numa tabela e num ficheiro.

PESQUISA SEQUENCIAL

Uma das operações mais frequentes de manipulação de dados é a pesquisa. Atualmente, e graças à massificação e maior poder de cálculo dos computadores pessoais, é possível fazer pesquisas rápidas de informação, pesquisas essas que outrora eram complexas ou demoradas de fazer.

A pesquisa sequencial (ou linear) é muitas vezes o único método de pesquisa disponível quando os dados a procurar não se encontram ordenados. Pesquisar sequencialmente um conjunto de dados corresponde a comparar um termo de pesquisa com cada um dos elementos do conjunto de dados, até se encontrar uma correspondência entre o termo de pesquisa e um dos dados do conjunto. A principal desvantagem da pesquisa sequencial é que esta obriga a percorrer todo o conjunto de dados para se encontrar todas as ocorrências do termo de pesquisa, por exemplo, todas as pessoas cujo primeiro nome é Carlos. Estes dados podem estar armazenados em tabelas ou em ficheiros. Numa pesquisa sequencial é necessário percorrer todo o conjunto de dados para verificar que o elemento a pesquisar não existe.

Vejamos um caso concreto: imaginemos que queremos procurar uma ficha numa tabela desordenada de fichas (*array* de *structs*), que tenha o campo nome igual a um determinado nome introduzido pelo utilizador:

```
#include <cstring>  // strcmpi()
// strcmpi(S1,S2); -> compara S1 com S2 ignorando se as letras
//                    estão em maiúsculas ou minúsculas.

const int MAX_FICHAS=10000;

typedef struct {   // definição da estrutura de cada ficha na tabela
  long int numBI;  // pode ter outros campos, não relevante para o exemplo
  char nome[151];
} TBI;
// continua na página seguinte
```

```
TBI Tab[MAX_FICHAS]; // tabela pode ter até MAX_FICHAS fichas do tipo TBI
                     // (não deve ser var global)
//-------------------------------------------------------------------------
// Função que devolve a posição, na tabela, da ficha que contém o nome igual
// ao nome na string de pesquisa 'nomePesquisar'. Procura desde a posição
// 'inicio' da tabela até à posição 'fim'.
// Devolve -1 se não encontrar nenhuma ficha.
int procuraNome(TBI tab[], int inicio, int fim, char *nomePesquisar) {
  for (int i=inicio; i<= fim; i++) {      // Um ciclo desde inicio até fim
    if ( strcmpi(nomePesquisar, tab[i].nome) == 0) // se encontrou a ficha,
      return i;                                    // então devolve índice
  }
  return -1;  // não encontrou nenhuma ficha com o mesmo nome, devolve -1
} // fim da função
//-------------------------------------------------------------------------
```

Como foi referido anteriormente a desvantagem da pesquisa sequencial reside no facto de ser necessário percorrer *toda* a tabela para nos certificarmos de que o valor que andamos à procura não existe.

A pesquisa do exemplo anterior podia ser feita em alternativa através do número do BI ou por qualquer outro dado que pertença à estrutura de dados. O exemplo seguinte ilustra uma pesquisa sequencial pelo número do BI:

```
//-------------------------------------------------------------------------
// Função que devolve a posição (índice) da ficha com número de
// BI igual a 'numBI' ou -1 caso não exista uma ficha com esse número de BI
int ProcuraBI(TBI tab[], int inicio, int fim, long int numBI) {
      for (int i=inicio; i<= fim; i++) {  // Um ciclo desde inicio até fim
            if ( numBI == tab[i].numBI )  // se encontrou a ficha, então
                  return i;               // devolve índice
      }
      return -1; //não encontrou nenhuma ficha com o número numBI: devolve -1
} // fim da função
//-------------------------------------------------------------------------
```

Como estimar o tempo de pesquisa sequencial a uma tabela de dados?

Supondo que se quer pesquisar um ficheiro (por exemplo) com **N** fichas e que demoramos um tempo de **t** segundos a "ler e verificar" cada ficha, então será de esperar que o tempo para achar um elemento no ficheiro (supondo que ele existe) seja, em média, igual a **t×N/2** segundos, supondo que a ficha a procurar pode estar numa qualquer posição do ficheiro. Diz-se então que a pesquisa sequencial ou linear tem um peso computacional proporcional ao número de dados a pesquisar, logo tem complexidade de ordem **N**, ou **O(N)**.

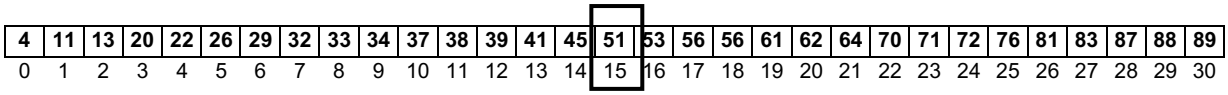
PESQUISA BINÁRIA

Quando o conjunto dos dados a pesquisar está ordenado segundo um determinado critério então podemos utilizar um método de pesquisa mais eficiente do que a pesquisa sequencial: a pesquisa binária, também conhecida por pesquisa dicotómica. Esta forma de pesquisa é bastante rápida e consiste no seguinte:

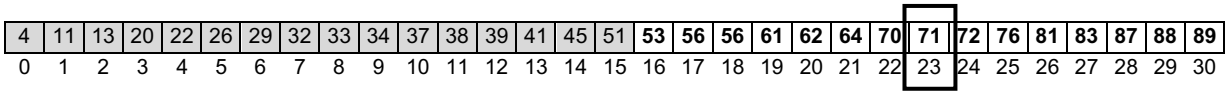
1. Toma-se o conjunto de dados ordenados a pesquisar;

2. Compara-se o elemento no meio do conjunto com o termo de pesquisa. Se forem iguais terminou a pesquisa;
3. Se o termo de pesquisa for anterior ao elemento do meio então reduz-se o conjunto a pesquisar apenas aos elementos anteriores ao elemento do meio. Se o termo de pesquisa for posterior ao elemento do meio então reduz-se o conjunto de dados a pesquisar à metade posterior ao elemento do meio.
4. Enquanto o conjunto resultante não for nulo, volta-se ao passo 2, agora com metade dos elementos para procurar.

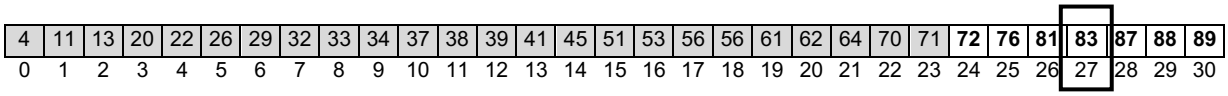
Vejamos um exemplo do funcionamento da pesquisa binária: Consideremos uma tabela de inteiros **tab** com 31 valores ordenados. Queremos saber em que posição da tabela está o número 81.



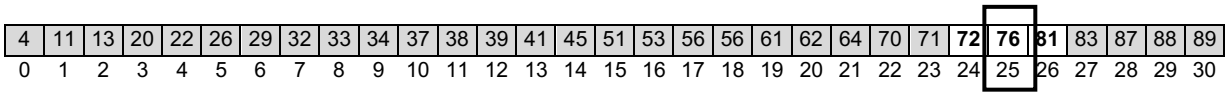
Começa-se a pesquisa pelo elemento do meio da tabela. Define-se o elemento do meio através da seguinte expressão: **(indiceFinal + indiceInicial)/2**. Neste caso concreto: **meio = (0 + 30)/2 = 15**. Como **tab[15] = 51 < 81**, então o elemento a pesquisar está entre a posição 16 e a posição 30 (ver abaixo):



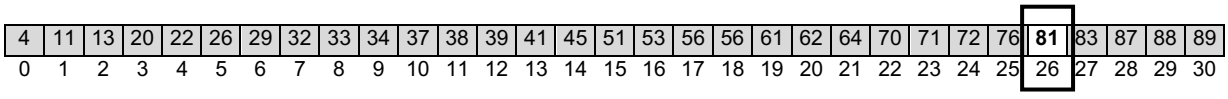
A pesquisa continua agora apenas na sub-tabela entre os índices 16 e 30. O novo elemento do meio é agora: **meio = (16+30)/2 = 23**. Como **tab[23] = 71 < 81**, o elemento a procurar está entre a posição 24 e 30.



A pesquisa continua agora na sub-tabela entre os índices 24 e 30. Mais uma vez é calculado o elemento do meio: **meio = (24+30)/2 = 27**. Como **tab[27] = 83 > 81** então o elemento a procurar está abaixo do elemento do meio, logo da posição 24 à posição 26.



A pesquisa continua agora entre os elementos com índices 24 a 26. O novo elemento do meio passa a ser: **meio = (24+26)/2 = 25**. Sendo t**ab[25] = 76 < 81**, o elemento a procurar está acima do meio, logo apenas poderá estar no elemento com índice 26.



Finalmente analisa-se o único elemento que falta: **meio= (26+26)/2 = 26**. Como **tab[26] = 81** então encontrou-se na posição 26 o elemento que se procurava. Foram apenas necessárias 5 iterações até se encontrar este valor. Se tivesse sido utilizada a pesquisa sequencial neste mesmo exemplo, teriam sido necessárias 27 iterações.

Resumindo, a pesquisa binária vai dividindo ao meio o conjunto a procurar até se chegar a um conjunto com apenas 1 elemento.

Como estimar o tempo de pesquisa binária de uma base de dados?

Supondo que se quer pesquisar um ficheiro (por exemplo) com **N** fichas e que demoramos um tempo de **t** segundos a ler e verificar cada ficha, então podemos esperar que o tempo necessário para encontrar um elemento no ficheiro (caso exista ou não) seja, em média, $\mathbf{t \times log_2(N+1)}$ segundos (supondo que o ficheiro se encontra ordenado e que a ficha a procurar se pode encontrar numa posição qualquer do ficheiro). Diz-se então que a pesquisa binária tem um peso computacional proporcional ao logaritmo dos dados a pesquisar, logo uma complexidade de ordem **log(N)**, ou simplesmente **O(log(N))**.

Na tabela abaixo apresenta-se a correspondência entre o número de elementos de uma tabela ordenada e o número de leituras necessárias até se encontrar o elemento utilizando pesquisa binária:

Número de elementos na tabela	Número de leituras (pesquisa binária)
16...31	5
1.000	10
100.000	17
1.000.000	20
100.000.000	27

Isto quer dizer que se tivermos um ficheiro com 100 milhões de dados ordenados, apenas temos que ler 27 desses dados até chegar à posição do dado procurado. Numa pesquisa sequencial teríamos que ler em média 50 milhões de fichas nessa mesma pesquisa. Podemos concluir que o facto de termos uma tabela ordenada permite reduzir em muito o tempo de pesquisa caso seja utilizada a pesquisa binária.

Implementação da pesquisa binária em C++

Podemos implementar a pesquisa binária utilizando dois tipos de abordagem: uma abordagem iterativa ou uma abordagem recursiva. A título de exemplo, consideremos uma tabela de **n** inteiros ordenados por ordem crescente. Consideremos em primeiro lugar a solução iterativa. O algoritmo de pesquisa binária pode ser resumido da seguinte forma:

Algoritmo de pesquisa binária:

Dados: **`inicio=0`**, o índice do primeiro elemento da tabela, **`fim = n-1`** o índice do último elemento da tabela e **`pesq`** o elemento de pesquisa;

Passo 1: Determina o elemento no meio da tabela **`meio = (inicio+fim)/2`**, arredondado para baixo;

Passo 2: Determina o valor da tabela no índice **`meio: valor=tab[meio]`**;

Passo 3: Se o valor a pesquisar **`pesq`** for igual a **`valor`** então foi encontrado o valor de pesquisa. Termina o algoritmo devolvendo o índice do **`valor`** (na posição cujo índice é **`meio`**).

Passo 4: Se **`pesq`** for maior do que **`valor`** então procura na sub-tabela da direita, ou seja o novo **`inicio=meio+1`**. Caso contrário procura na sub-tabela da esquerda, ou seja o novo **`fim=meio-1`**;

Passo 5: Se **`inicio <= fim`** então repete de novo a partir do passo 1;

Passo 6: Como **`inicio > fim`** ==> Insucesso! Não foi encontrado qualquer elemento igual na tabela.

Com base neste algoritmo, podemos chegar à seguinte implementação da pesquisa binária:

```
//Função que devolve o índice da tabela de inteiros 'tab' que contém o número
//'valor'. A tabela tem 'comp' elementos.
// No caso de não existir, a função devolve -1
int pesquisaBinaria(int tab[ ], int comp, int valor) {
  int inicio = 0;       // inicio tem o índice do primeiro elemento da tabela
  int fim = comp - 1;   // e fim tem o índice do último elemento
  while (inicio <= fim){  // Enquanto a tabela a pesquisar tiver elementos
    int meio = (inicio+fim)/2;  // determina o índice do elemento do meio
    if (valor == tab[meio]) return meio; // O valor pretendido foi encontrado
    else
      if (valor > tab[meio])  // O valor é maior?
        inicio = meio + 1;    // Então procura na sub-tabela da direita
      else                    // Senão (o valor é menor e)
        fim = meio - 1;       // procura na sub-tabela da esquerda.
  }
  return -1;  // O valor pretendido não foi encontrado
}  // Fim da função
```

A solução recursiva é em tudo idêntica à resolução iterativa. A condição de paragem é a tabela ser vazia. Neste caso, não é possível encontrar o valor e a função devolve -1. No caso geral, compara-se o valor a procurar com o valor no meio da tabela. Se o valor for maior então devolve-se (de uma forma recursiva) o resultado da pesquisa na sub-tabela da direita. Se o valor for menor então devolve-se (também de uma forma recursiva) o resultado da pesquisa na sub-tabela da esquerda. No caso de o valor ser igual ao elemento do meio então devolve-se o índice do elemento do meio. A função de pesquisa passa a ser:

```
// Função de pesquisa recursiva que devolve o índice da tabela 'tab'
// que contém o número 'valor'.
// A tabela é definida pelo índice inicial 'inicio' e pelo índice
// final 'fim'. No caso de não existir o valor a função devolve -1.
int pesquisaBinariaRec(int tab[ ], int inicio, int fim, int valor) {
  if(inicio>fim) return -1;   //A tabela está vazia - valor não encontrado
  int meio = (inicio+fim)/2;    // Calcula o índice do elemento do meio
  if(valor>tab[meio]) //O valor é maior então procura na sub-tabela à direita
    return(PesquisaBinariaRec(tab, meio+1, fim, valor));
  if(valor<tab[meio]) //O valor é menor então procura na sub-tabela à esq.
    return(PesquisaBinariaRec(tab, inicio, meio-1, valor));
  return meio;    // Se não maior nem menor então só pode ser igual…
} // Fim da função
```

De notar que o protótipo da função passa a ser diferente uma vez que necessitamos de indicar nas sucessivas chamadas à função os limites de cada uma das sub-tabelas. Nesse sentido, deixamos de passar o comprimento da tabela para passar a indicar o índice inicial e o índice final.

PESQUISAS NUM FICHEIRO DE FICHAS

Escrever um programa para fazer pesquisa sequencial num ficheiro é relativamente simples. O procedimento é relativamente básico: abrir o ficheiro; ler e comparar cada ficha com o termo de pesquisa, até se encontrar uma ficha com um campo de pesquisa igual ao termo de pesquisa, ou até se atingir o fim

do ficheiro. Neste último caso, deve ser dada uma indicação especial para assinalar que a ficha pretendida não foi encontrada. Finalmente dever-se-á sempre fechar o ficheiro. **Deixa-se ao cuidado do aluno implementar um código de exemplo.**

Pesquisa binária num ficheiro binário

Para se fazer uma pesquisa binária (dicotómica) num ficheiro binário contendo fichas, o procedimento é análogo ao utilizado na pesquisa binária numa tabela. Há no entanto algumas diferenças resultantes das particularidades dos ficheiros: é preciso analisar quantas fichas estão guardadas em ficheiro antes de se iniciar a pesquisa e é necessário ler uma ficha do ficheiro antes de ela poder ser analisada.

A seguir apresenta-se um exemplo de uma função para fazer uma pesquisa binária num ficheiro de fichas com informação sobre o BI. Estas fichas estão ordenadas pelo campo altura.

```
typedef struct {      // definição da estrutura de cada ficha no ficheiro
      long int numBI; // pode ter outros campos, não relevante para o exemplo
      char nome[151];
      double altura;
} TBI;
//-----------------------------------------------------------------------
// Função que devolve a posição da ficha com altura igual ao
// parâmetro 'altura' ou -1 no caso de não existir nenhuma ficha
// com a altura procurada. Esta função implementa a pesquisa
// binária no ficheiro dado por 'nomeFicheiro'. Para além disso,
// a função devolve no parâmetro 'ficha' a ficha retirada do ficheiro.
int pesquisaBinariaFicheiro(char *nomeFicheiro, float altura, TBI *ficha) {
  int inicio, meio, fim, n; // variáveis auxiliares
  // abre ficheiro apenas se ele já existir
  ifstream fic(nomeFicheiro, ios::binary);
  if (!fic.is_open() ) return -1; // Não foi possível abrir o ficheiro, sai!
  // Se chegou aqui então foi porque conseguiu abrir o ficheiro para leitura
  fic.seekg(0, ios::end);   // Aponta para o elemento 0 a partir da última
                            // posição, ou seja aponta para o último elemento
  n = fic.tellg()/sizeof(TBI);
  // O método tellg() devolve a posição em bytes do próximo elemento que se
  // vai ler do ficheiro. Como o ponteiro de leitura está a apontar para o
  // fim do ficheiro, este devolve o número de bytes que o ficheiro possui.
  // n passa a ter o número de fichas existentes no ficheiro
  inicio = 0;  // define o inicio do conjunto de pesquisa
  fim = n-1;   // define o fim do conjunto de pesquisa
  while ( inicio <= fim ) {
    meio = (inicio+fim)/2;  // meio <- posição do elemento no meio da tabela
    fic.seekg(meio*sizeof(TBI)); // Aponta para a posição do ficheiro onde
                                 // está a ficha no meio do conjunto de pesquisa


// continua na próxima página…
```

```
    // lê a ficha para memória. Uma ficha ocupa exactamente sizof(TBI) bytes
    fic.read( (unsigned char *)ficha, sizeof(TBI));
    if (altura == ficha->altura) {  // encontrada ficha com altura procurada
      fic.close(); // Fecha o ficheiro, pois terminaram as leituras.
      return meio; // A estrutura ficha tem os dados da ficha e a função deve
    }              // devolver a posição do ficheiro onde estava a ficha.
    if (altura > ficha->altura) // Se altura > altura no meio da tabela
      inicio = meio + 1;        // Então procura na 2ª metade do sub-ficheiro
    else                        // Senão (altura é menor e)
      fim = meio - 1;           // procura na 1ª metade do sub-ficheiro
  } // fim do while
  fic.close();    // Fecha ficheiro pois terminaram as leituras
  return -1;      // Não encontrou qualquer ficha com a altura pretendida
}  // Fim da função
```

PESQUISA EM FICHEIRO DE TEXTO

Fazer uma pesquisa sequencial num ficheiro de texto é relativamente fácil. O processo consiste em ler o ficheiro linha a linha, comparando cada linha com a *string* a procurar. **Deixa-se ao cuidado do aluno implementar um exemplo.**

Uma pesquisa binária num ficheiro de texto torna-se numa tarefa mais complexa, pois uma linha de texto, ao contrário de uma ficha (`struct`) não ocupa um número fixo de bytes. Ora, apontar para o início de uma linha no meio de um ficheiro obriga normalmente a uma procura sequencial do sítio onde essa linha começa e acaba. Apesar deste detalhe, a pesquisa binária pode também ser aplicável a ficheiros de texto, embora a forma de o conseguir não seja o objetivo desta unidade curricular.

EXERCÍCIOS SUGERIDOS

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à pesquisa de dados. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas.

Problema 5.1 – Pesquisa sequencial de números – Fácil

Escreva uma função que permita procurar, numa tabela contendo **N** números inteiros, a primeira ocorrência de um determinado número. A função deverá receber como parâmetros um ponteiro para a tabela, o número de elementos da tabela e o número a procurar. Deverá devolver o índice do elemento da tabela onde se encontra o valor ou –1 caso este não exista na tabela.

Teste a função com um programa feito por si e com os valores da tabela a serem inicializados aleatoriamente (considere N=100 e que cada elemento pode variar entre –10 e 42).

Problema 5.2 – Pesquisa sequencial de uma substring – Fácil

Escreva uma função que permita procurar, numa tabela com **N** *strings* (com até 100 caracteres), a primeira ocorrência de uma determinada *substring*. A função deverá receber como parâmetros um ponteiro para a tabela de *strings*, o número de elementos que contém a tabela e um ponteiro para a *substring* que se pretende procurar. A função deverá devolver o índice do 1º elemento da tabela onde aparece a *substring* ou então –1, se não foi encontrada a *substring* na tabela. Abaixo está um possível protótipo da função:

```
int procuraSubString(char tab[][101], int numElemsTab, char *subString);
```

Sugestão: Utilize a função `strstr()` para o ajudar na resolução deste problema.
Teste a função num programa feito por si e com 6 valores na tabela inicializados como constantes.

Problema 5.3 – Pesquisa binária de número – Médio

Implemente uma função que devolva quantos valores de uma tabela ordenada de *doubles* (temperaturas máximas anuais) estão acima de um dado valor **`min`**. A função deverá usar pesquisa binária para descobrir a posição na tabela do valor a seguir ao procurado. Os parâmetros de entrada da função deverão ser: um ponteiro para a tabela, o número de elementos na tabela e o valor **`min`**. A função deverá devolver o número de valores na tabela com valor acima do valor **`min`**.

Nota: ao contrário das funções normais de pesquisa binária, esta função não devolve –1 quando o valor não existe. O que se quer devolver é o número de valores na tabela acima do valor **`min`** e portanto, mesmo que o valor não exista na tabela a função pode devolver valores de 0 até n. Também pode suceder existirem vários elementos na tabela com o valor procurado.

Teste a função com um programa feito por si e com os n valores da tabela a serem inicializados aleatoriamente (considere n=100 e que cada elemento pode variar entre –12.5 e 45.5).

Problema 5.4 – Pesquisa binária (e inserção eventual) de inteiro – Médio

Implemente uma função que insira um número inteiro (n.º de sócio) numa tabela ordenada de inteiros, se este não existir na tabela. A função deverá usar pesquisa binária para procurar na tabela o número procurado. Se este não existir, insere-o na posição correta de forma a manter a tabela ordenada.

Os parâmetros de entrada da função deverão ser: um ponteiro para a tabela, o número de elementos na tabela, o número máximo de elementos da tabela e o número inteiro a pesquisar (e a inserir eventualmente). A função deverá devolver zero se o número já existir, 1 no caso de ele ser inserido e –1 se o número não existir mas a tabela estiver cheia.

Teste a função com um programa feito por si e com 10 valores da tabela a serem inicializados aleatoriamente. Considere que o nº máximo de elementos na tabela é 15 e que cada elemento pode variar entre 1 e 100.

Problema 5.5 – Pesquisa binária de tabela indexada – Médio/Demorado

Faça um programa que permita fazer uma pesquisa binária a uma tabela indexada de *structs*. Para facilitar a tarefa defina uma tabela desordenada de BI's com 15 fichas (nome, número, altura, naturalidade) com 3 tabelas de indexação, uma que possui a ordenação por nome, outra por número e outra por altura. O programa deverá ter várias opções: 1-procurar por nome, 2-procurar por número 3-procurar por altura, 4-nome seguinte, 5-número seguinte, 6-altura seguinte. Todas as opções de procura

devem implementar pesquisa binária à tabela indexada. Inicialmente, antes da primeira pesquisa, a "ficha seguinte" será a primeira da sua ordem.

Problema 5.6 – Pesquisa binária de nome – Difícil/Demorado

Faça uma função que imprima no monitor todas as fichas de uma tabela de *structs* ordenada por nome, que possuam o nome de uma pessoa começado por uma substring passada como parâmetro. A função deverá usar pesquisa binária. Um possível protótipo para esta função será:

```
void imprimeInicio(TBI tab[], int nelems, char *subString);
```

A estrutura de cada ficha e a tabela poderão ser:

```
typedef struct {  // local onde está definido o tipo TBI, uma ficha com os
  char nome[30];  // dados de um bilhete de identidade
  long int numBI;
  float altura;
} TBI;

TBI tabBI[100];   // sem ser variável global
```

Um possível resultado para `imprimeInicio(tabBI, 50, "Carlos");` seria:

```
[20]: 8123457,               Carlos Alberto, 1.70 m
[21]:11828399,          Carlos Luis Antunes, 1.80 m
[22]: 9327681,                 Carlos Silva, 1.74 m
```

Notas:

- Utilize a função **`strcasecmp()`** ou **`strcmpi()`** para o ajudar na resolução do problema. Estas funções fazem o mesmo que a função **`strcmp()`** mas não fazem distinção entre letras minúsculas e letras maiúsculas.
- Uma pesquisa binária não acha obrigatoriamente a primeira ocorrência de uma dada igualdade, mas apenas uma delas. Na resolução deste problema, é preciso ter isto em conta.

Problema 5.7 – Comparação entre pesquisa sequencial e binária – Médio/Demorado

Faça um programa que permita comparar o desempenho entre uma pesquisa sequencial (ou linear) e uma binária (ou dicotómica) a um ficheiro com 1 milhão de inteiros.

O programa deverá ter 3 opções num menu:

1 – Criar o ficheiro: deverá criar o ficheiro com 1 milhão de inteiros múltiplos de 10, a começar no 10 e a terminar no 10.000.000 (pretende-se criar um ficheiro conhecido);

2 – Procurar a posição de um inteiro por pesquisa sequencial. Pede-se um número e mostra-se a sua posição (índice) no ficheiro. Por exemplo 10000000 daria 999999, 7 dará –1 (nº inexistente), e 10 dará 0. Esta opção deverá também imprimir no ecrã quanto tempo demorou a encontrar a posição.

3 – Procurar a posição de um inteiro por pesquisa binária. O comportamento deverá ser análogo ao da opção 2. Esta opção deverá mostrar no ecrã o tempo que demorou a pesquisa.

Problema 5.8 – Pesquisa sequencial – Médio (saiu no exame da época de recurso de 05/07/2016)

Considere uma tabela de fichas (`structs`) de filiais de uma empresa, sendo cada ficha constituída por: nome (55 caracteres úteis), nº de trabalhadores (inteiro) e volume de negócios (real de precisão dupla, em euros). A tabela pode conter até 200 fichas, havendo uma variável (`nfiliais`) com o nº de fichas existentes na tabela.

a) Declare a `struct`.

b) Implemente uma função para contar (e devolver) o nº de sucursais com volume de negócios superior a `valorInf` e inferior a `valorSup`. A função, chamada `contaFiliais()`, tem como parâmetros uma tabela de elementos do tipo declarado na alínea a), o nº de elementos (filiais) na tabela, `valorInf` e `valorSup`. Esta função não escreve nada no ecrã.

c) Implemente uma função `main()` de teste para mostrar no ecrã o nº de filiais com volume de negócios superior a 100.000 euros e inferior a 300.000 euros, para uma tabela inicializada por si com 4 filiais.

Problema 5.9 – Pesquisa sequencial – Médio (adaptado do 1º teste de frequência de 20/03/2012)

Considere uma tabela de fichas (`structs`) de alunos de uma disciplina, sendo cada ficha constituída por nome (50 caracteres úteis), nº de aluno e ano de nascimento.

a) Declare a `struct`.

b) Faça uma função para eliminar de uma tabela de fichas do tipo definido em a), passada como parâmetro da função, uma ficha dada pelo seu índice (também passado como parâmetro da função). O nº de elementos da tabela é passado por referência. A função não precisa de verificar a consistência dos parâmetros.

c) Faça uma função para eliminar de uma tabela destas, passada como parâmetro da função, a ficha da pessoa mais velha (se houver várias pessoas mais velhas com a mesma idade pode ser eliminada uma qualquer).

d) Implemente uma função `main()` de teste, para uma tabela inicializada com 5 alunos.

Obs.: Aconselha-se a elaborar previamente em papel um esboço ou desenho da tabela, de modo a perceber o problema e as suas implicações. Comente o código.

Problema 5.10 – Pesquisa binária – Médio (adaptado do 1º teste de frequência de 29/03/2017)

Considere a estrutura de dados: `struct ficha{ int id; float campo_1; float campo_2; };`
Implemente a função recursiva `int pesquisaBinaria(ifstream &ficheiro, int inicio, int fim, float valor)` para realizar uma pesquisa binária de `valor` no campo `campo_2` das fichas armazenadas no ficheiro em modo binário. Pode assumir que o ficheiro já se encontra aberto e que as fichas estão ordenadas por ordem crescente segundo os valores guardados em `campo_2`. A função deve devolver a posição no ficheiro da primeira ficha encontrada contendo `valor` no campo `campo_2`, entre os índices `inicio` e `fim` do ficheiro (posições absolutas a partir do início do ficheiro), ou -1 se não for encontrada a ficha pretendida.
Recorde que `ficheiro.seekg(pos)` ajusta o ponteiro para leitura na posição absoluta `pos` a partir do início do ficheiro e que a função `ficheiro.read(unsigned char *buffer, int n)` lê em modo binário `n` bytes de informação do ficheiro, guardando a informação lida na tabela `buffer`.

Problema 5.11 – Pesquisa sequencial – Fácil (adaptado do 1º teste de frequência de 12/03/2014)

Considere uma tabela de fichas (`structs`) de sucursais de uma empresa, sendo cada ficha constituída por: `nome` (60 carateres úteis), `empregados` (inteiro) e `area` (real, em m^2).

a) Declare a `struct`.

b) Implemente uma função para retornar o número de sucursais com área superior a `areaDada`. A função, chamada `contaDim()`, tem como parâmetros uma tabela de elementos do tipo declarado na alínea a), o n.º de elementos (ou sucursais) na tabela e `areaDada`. Considere que a tabela está ordenada por área crescente, não precisando por esse motivo de fazer uma pesquisa sequencial completa da tabela.

c) Implemente uma função `main()` de teste para contar e mostrar no ecrã o número de sucursais com área superior a 100 m^2, para uma tabela inicializada com 5 sucursais.

Obs.: Comente o código. Os 100 m^2 devem ser definidos numa constante global.

Problema 5.12 – Pesquisa binária de tabela indexada – Médio

Faça um programa que permita fazer uma pesquisa binária a uma tabela indexada de fichas (`structs`). Para facilitar a tarefa defina uma tabela ordenada por ordem alfabética de cartões de cidadão (CC) com 15 fichas (nome, número, altura, naturalidade) e uma tabela de indexação para a ordenação por número. O programa deverá previamente efetuar a ordenação da tabela de índices, por número crescente do CC. O programa deve solicitar repetidamente ao utilizador um número de CC a procurar, mostrando os vários campos da ficha no caso de o número existir e uma mensagem adequada no caso contrário. O programa termina quando o utilizador introduzir o número 0 (zero).